The SLA Model

Keven T. Kearney and Francesco Torelli

Abstract This chapter describes the SLA model that has been developed by the SLA@SOI project. It defines a syntax for machine-readable Service Level Agreements (SLAs) and SLA templates (SLA(T)). Historically, the SLA was developed as a generalisation and refinement of the web service-specific XML standards: WS-Agreement, WSLA, and WSDL. Instead of web services, however, the SLA model deals with services in general, and instead of XML, it is language independent. The SLA model provides a specification of SLA(T) content at a fine-grained level of detail, which is both richly expressive and inherently extensible: supporting controlled customisation to arbitrary domain-specific requirements. The model has been applied to a range of industrial use-cases, including: ERP hosting, Enterprise IT, live-media streaming, and health-care provision. At the time of writing, the abstract syntax has been realised in concrete form as a Java API, XML-Schema, and BNF Grammar.

1 Introduction

This chapter describes the SLA Model employed within SLA@SOI. An SLA is an agreement between a service provider and service customer about the required quality-of-service (QoS) characteristics of some service(s) delivered by the provider to the customer. Properly speaking, the agreement *as such* is the intangible understanding, or accord, that exists between the provider and customer. The SLA Model is not concerned with the intentional aspects of an agreement. It is only concerned

Keven T. Kearney
Engineering Ingegneria Informatica spa, Via Riccardo Morandi, 32, 00148 Roma, Italy,
e-mail: keven.kearney@eng.it

Francesco Torelli
Engineering Ingegneria Informatica spa, Via Riccardo Morandi, 32, 00148 Roma, Italy,
e-mail: francesco.torelli@eng.it

with modelling the *physical* document that serves as the formal, concrete representation of an agreement. The SLA Model is therefore a *document model*. In particular, it is an *abstract syntax*, specifying, in a language independent manner, the formal *serialised content* of SLA and SLA Template (SLAT) documents. The SLA Model assumes the basic domain concepts of 'SLA', 'SLA Template' and 'service' described elsewhere in this book, but it is *not* a conceptual model of this domain. For present purposes, an SLA is a document, the formal syntactic content of which is specified in an abstract way by the SLA Model presented here.

The key objective in developing the SLA Model is to meet two ostensibly conflicting requirements of SLA@SOI: On one side, the model needs to support the generic capabilities encapsulated by the 'Generic SLA Manager' (GSLAM; described in Chapter 'GSLAM – The Anatomy of the Generic SLA Manager'), requiring that QoS guarantees and party obligations be specified in a domain-*independent* manner at a fine-grained level of detail. On the other side, however, in order to meet the diverse domain-*specific* requirements of the SLA@SOI test-bed scenarios and real-world applications, the model must remain open to extension and customisation.

The domain-*independent* operations of the GSLAM span the entire SLA lifecycle and include:

- quality-of-service (QoS) based service discovery,
- SLA negotiation, planning and optimisation of service delivery systems to achieve the goals expressed in SLAs, and
- the subsequent monitoring and potential modification of these systems to ensure these goals are indeed satisfied.

While complex and diverse, these capabilities can all, in essence, be characterised as entailing some form of multiple constraint satisfaction, where the constraints to be satisfied are the QoS guarantees expressed in SLAs. Accordingly, the SLA Model must provide a common, domain-independent means for the detailed and precise expression of these *constraints*. At the same time, however, it is impossible to foresee and enumerate all the possible requirements of domain-*specific* applications. Thus the SLA Model also needs to support the definition and expression of custom constraints.

To meet these conflicting requirements, the SLA Model is designed as a domain-independent model of SLA(T)[1] content grounded in an *abstract constraint language*, the concrete elements of which are formally specified by 'plug-in' domain-specific vocabularies. The constraint language provides a consistent, fine-grained language supporting operations research, while the vocabularies provide for controlled extensibility.

Historically, the SLA Model has been developed as a generalisation and refinement of the web-service specific XML standards WS-Agreement [1], WSLA [2], and WSDL [3] abstracting the notion of 'web-service' to the more generic 'service', and eliminating the unnecessary restriction to XML as a representational format. The SLA Model thus supports the formulation of SLAs in *any* language for

[1] We use the acronym SLA(T) to refer collectively to SLAs and SLATs

any service. To support as wide a range of domain-specific scenarios as possible, the SLA Model only specifies the *minimal* content of SLA(T)s, encapsulating only those aspects of SLA(T)s necessary for the generic functions of the GSLAM.

This chapter is organised as follows: Section 2 introduces the basic modelling approach and provides foundational definitions. Sections 3 to 6 then describe, in order, the content of SLA(T)s, service interface specifications, the abstract constraint language, and domain-specific vocabularies. Section 7 closes the chapter with a detailed walk-through of an example SLA Template represented in concrete XML syntax.

2 Basic Concepts

The SLA Model is an abstract syntax specifying the formal content of serialised SLA(T) documents. In purely formal terms, we define a document in generic terms as *an hierarchical organisation of symbols*. This book, for example, is a document comprised of a sequence of letters and punctuation marks (the symbols), hierarchically divided into chapters, sections, subsections, paragraphs, and so on. The SLA Model is a *syntax* because it serves to specify the *organisation of symbols* in an SLA document, but it is *abstract* in that it leaves unspecified the particular symbols used to instantiate this organisation.

We will refer to any organisation of symbols as an *expression* (this sentence, for example, is an expression), and to classes of expressions as *expression-types*. The SLA Model is specified in terms of expression-types. Formally, we treat each expression-type as a *set* whose members are the expressions which instantiate that type. In the remainder of this chapter, we will use the terms *type* and *expression-type* interchangeably. To avoid ambiguity, we will also use *token* (meaning an 'instance of a type') as synonymous with *expression*.

The SLA Model also draws a distinction between tokens *per se* and tokens that are *references to tokens*. Specifically, if **T** is an expression-type, then:

- **T** : denotes the set of tokens of type **T**,
- $\uparrow$**T** : denotes the set of references to tokens of type **T**,
- $\Uparrow$**T** : denotes the set of references to (subtypes of) type **T**, while
- **(**$\uparrow$**)T** : denotes either a token, or a reference to a token, of type **T**.

The universal type, which is the set of all possible expressions, is denoted by the symbol **L*** (where 'L' may be read as 'legal expressions', or simply 'language'). The asterix is used here, and in subsequent type names, to indicate that the type is abstract (meaning that it cannot be directly instantiated).

To capture the hierarchical organisation of documents, we introduce a first high-level expression-type, **E*** $\subset$ **L***, denoting a class of *entity expressions*, each token of which is just a collection of key/value attribute pairs. Attribute values can be any kind of expression, including other entity expressions, which thus permits the hierarchical nesting of entities. Formally:

<u>**E***</u> $\subset$ **V*** : each token is an unordered collection of ordered key/value attribute pairs <***k*,*v***>, where: ***k*** $\in$ **NAME** is the name (key) of the attribute, and ***v*** $\in$ **L*** gives the attribute's value.

The **NAME** type referred to in this definition is a *datatype*, in this case denoting a class of *simple names*. All datatypes belong to a second high-level expression-type, **V*** $\subset$ **L***, denoting a class of *value* expressions. In particular, a datatype is a specialisation of a generic **Constant*** $\subset$ **V*** type denoting constant values (e.g. Boolean values ('true' or 'false') or metric quantities ('4 s', '10 bytes', etc.), web and e-mail addresses, and so on). We will describe **V***, and explain datatypes in more detail, in Section 5. Additional datatypes will be introduced in the text as the need arises.

Every document is an entity expression, that is, a token of type **E***. From the definition above, this means that a document is just an ordered collection of key/value attributes. In order to specify a document, therefore, we need to specify the particular *entity-expression-types* — i.e. subtypes of **E***, henceforth just 'entity-type' — from which the document is composed. As a first step, we introduce a generic *document*-type, encapsulating the common attributes of all documents. We define this document-type as follows (the notation is explained below):

<u>**Document***</u> $\subset$ E*
  vocabularies $\subset$ ($\uparrow$)Vocabulary $_{[0+]}$

The first line of this definition declares **Document*** as an expression-type that specialises (i.e. is a subset of) the type **E***. Each subsequent line defines a key/value attribute pair, or *attribute-type*. In this case, there is only one attribute-type, whose key is 'vocabularies' and whose value-type (denoted by the $\subset$ relation) is an array of 0+ (*zero or more*) **($\uparrow$)Vocabulary** expressions (i.e. either tokens or references to tokens of the type **Vocabulary**). The type **Vocabulary** is the generic entity-type for all vocabulary documents, and will be explained in detail in Section 6. For now, it suffices to state that a vocabulary is a collection of expression-type definitions. Semantically, the *vocabularies* attribute of a **Document*** token lists all the vocabularies required to specify, and hence validate, the content of that **Document*** token.

To specify an SLA, we will also require two further entity-types — **NamedEntity*** and **Macro** — which have the following definitions:

<u>**NamedEntity***</u> $\subset$ E*
  name $\subset$ NAME $_{[1]}$

<u>**Macro**</u> $\subset$ NamedEntity*
  expression $\subset$ V* $_{[1]}$

A **NamedEntity*** token is simply an entity expression which carries a single *name* attribute, the value of which can be used to refer to the token from other parts of the document. **Macro** inherits the *name* attribute from **NamedEntity***, and also carries a second *expression* attribute, whose value can be any token of type **V***. A **Macro** token serves a similar purpose to a **NamedEntity*** token, but this time its *name* value, when used as a reference, is always interpreted as referring to the value of the token's *expression* attribute, rather than the token itself. A **Macro** token with the *name* 'X' and *expression* 'abcdef', for example, permits the expression 'X' to be used in place of 'abcdef'. Macros are essentially a convenience feature, providing a means to decompose complex value expressions, and hence improve readability.

The next section builds on these basic definitions to specify the content of SLA(T) documents.

3 SLAs and SLA Templates

Historically, the high-level structure of an SLA(T), as defined by the SLA Model, has its roots in, and still maintains much in common with, WS-Agreement. Briefly, an SLAT is a document which comprises three sections, describing:

- the *parties* to the agreement,
- the relevant services, specified in terms of their functional *interfaces*, and
- the *agreement terms*, including *quality-of-service* (QoS) guarantees and other party obligations.

In formal terms, this document structure is captured by the following entity-type definition:

<u>**SLAT**</u> $\subset$ Document*
  parties $\subset$ Party $_{[2+]}$
  interfaceDeclrs $\subset$ InterfaceDeclr $_{[1+]}$
  agreementTerms $\subset$ AgreementTerm $_{[1+]}$
  macros $\subset$ Macro $_{[0+]}$

The types **Party**, **InterfaceDeclr** (interface declaration) and **AgreementTerm** are all entity-types which we will define formally in the subsections below.

The SLAT entity-type also includes an optional *macros* attribute. Although macros are essentially a convenience feature (as described earlier), their essentially symbolic (referential) properties can be exploited to serve more significant purposes. In SLA(T)s we exploit *macros* in order to encode customer *options*. This is done by introducing a **Macro** subtype, **Customisable**, defined as follows:

<u>**Customisable**</u> $\subset$ Macro
  domain $\subset$ Domain $_{[1]}$

The **Domain** type is part of the abstract constraint language and will be defined formally in Section 5.1. For now, it suffices to state that the *domain* attribute specifies a set of alternative values, with the value of the *expression* attribute then denoting a particular selection from these alternatives. In an SLAT this selection is interpreted as the 'default option', while in SLAs it is interpreted as the 'option chosen by the customer'. The example SLAT in Section 7 illustrates this use of the **Customisable** macro.

An SLA document has the same structure as an SLAT, but with additional attributes giving the time at which the SLA was agreed, its effective lifespan, and a reference to the template (if any) from which it was derived. To denote a specific point-in-time, or time-stamp (e.g. 'Wed Dec 15 18:38:0.0 CET'), we introduce a **DATETIME** datatype. For the template reference we use a reference type (as explained in the previous section), in this case ↑**SLAT**. An SLA document can then be modelled formally as:

$\underline{\textbf{SLA}} \subset \text{SLAT}$
 agreedAt $\subset$ DATETIME $_{[1]}$
 effectiveFrom $\subset$ DATETIME $_{[1]}$
 effectiveUntil $\subset$ DATETIME $_{[1]}$
 template $\subset \uparrow$SLAT $_{[0..1]}$

These two definitions completely capture the high-level structure of SLAs and SLATs. In the following subsections we move stepwise through the document hierarchy to specify SLA(T) content in more detail.

3.1 SLA(T) Parties

Information about a particular agreement party (e.g. the service provider, or the service customer) is captured using a **Party** entity-type, which is a concrete specialisation of a more generic **Actor*** entity-type. The relevant definitions are:

$\underline{\textbf{Actor*}} \subset \text{NamedEntity*}$

$\underline{\textbf{Party}} \subset \text{Actor*}$
 role $\subset$ ENUM $_{[1]}$
 operatives $\subset$ Operative $_{[0+]}$

The *role* attribute of **Party** serves to identify the role played by the party in the agreement. Typically, this role will be either 'service provider' or 'service customer', but there may be other roles peculiar to specific domains. Within any given domain, however, there will only be a handful of valid roles. As such, we need a mechanism by which we can state that the value of an attribute will be drawn from a limited set of *domain-specific* alternatives. The **ENUM** datatype, denoting an *enumerated* list, serves this purpose (the enumerated items themselves are specified in domain-specific vocabularies using **DataValue** tokens, described in Section 6).

Conceptually, each party to an SLA may be acting as an agent, or proxy, *on behalf of others*. A company executive, for example, can sign a contract for a catering service on behalf of the company's employees, who are the end consumers proper of the service. In the SLA Model, the individuals, if any, represented by a party are referred to as 'operatives'. A single SLA(T) may offer different QoS guarantees to different categories of operative, with each category described by an **Operative** entity-type expression:

Operative ⊂ Actor*

Note that both **Party** and **Operative** specialise the abstract **Actor*** type, and thus describe SLA 'actors'. As with all definitions in the SLA* model, the **Party** and **Operative** definitions are intended to capture only the minimal information and/or distinctions required to specify the agreement terms. It is expected that domain-specific vocabularies will extend these actor definitions to add more detailed information.

3.2 SLA(T) Interface Declarations

All information about the functional capabilities of a service is captured in the form of an **Interface** entity-type, a detailed description of which will be given later in Section 4. For the moment, it is sufficient to note that the **Interface** type essentially encapsulates the information found in traditional 'service descriptions' (in particular, WSDL documents). What is important in an SLA(T) is that all the relevant interfaces are *declared*, and this is achieved using an **InterfaceDeclr** entity-type, which has the following definition (the parent **Service*** type will also be defined in Section 4):

InterfaceDeclr ⊂ Service*
 provider ⊂ ↑Actor* $_{[1]}$
 consumers ⊂ ↑Actor* $_{[1+]}$
 endpoints ⊂ Endpoint $_{[1+]}$
 interface ⊂ (↑)Interface $_{[1]}$

Each **InterfaceDeclr** entry in an SLA(T) asserts an obligation on the part of one of the SLA(T) actors, as given by the *provider* attribute, to provide specific functional capabilities to one or more other actors, given by the *consumers* attribute. Note that both *provider* and *consumers* attributes accept references to *any* actor — i.e. to *any* **Party** or **Operative** — regardless of that actor's role in the agreement. It may be, for example, that we wish to oblige a service provider to send regular status reports to the service customer, a prerequisite for which is that the *customer* provides a suitable interface for receiving these reports.

In addition to specifying the relevant actors, each **InterfaceDeclr** also enumerates one or more *endpoints*, each of which provides a *location* (address) and a communications *protocol* by which interface operations may be invoked:

Endpoint ⊂ NamedEntity*

protocol $\subset$ ENUM $_{[1]}$
location $\subset$ TEXT $_{[0..1]}$

Just as with the **Party** *role* attribute (Section 3.1 above), the **ENUM** value-type defined for the *protocol* attribute indicates that values will be drawn from some limited set of *domain-specific* alternatives, such as 'SOAP', 'HTTP', 'e-mail', 'voice-telephony', 'SMS', and so on. The choice of protocol also determines the appropriate form for *location* values. For example, for 'e-mail', an e-mail address is required, while for 'voice-telephony', the location would be a telephone number. Accordingly, we define the value-type of the *location* attribute as **TEXT**, a datatype denoting some opaque string constant. Note that the *location* attribute is optional, since it is not necessarily the case that locations can be fixed in advance.

Finally, the interface that is the subject of the declaration is given by the *interface* attribute, whose value may be an embedded **Interface** document, or more typically, a reference to an **Interface** document accessible from some external source. Note that several endpoints may be defined for a single interface, and that the same interface may appear in multiple interface declarations.

3.3 SLA(T) Agreement Terms

The agreement terms section of an SLA(T) specifies the QoS guarantees and other party obligations that form the substantive content of the agreement. An SLA(T) may contain multiple agreement terms, each of which can define multiple guarantees effective under varying conditions. The **AgreementTerm** entity-type is defined as follows:

AgreementTerm $\subset$ NamedEntity*
pre $\subset$ Constraint* $_{[0..1]}$
macros $\subset$ Macro $_{[0+]}$
guarantees $\subset$ Guarantee* $_{[1+]}$

The *pre* attribute specifies (optional) *pre*-conditions on the agreement term, defining the conditions under which the agreement term is effective. (If none are given the agreement term is always effective.) These *pre*-conditions take the form of a **Constraint*** expression, which is part of the *abstract constraint language* and will be explained in detail in Section 5.1. The *macros* attribute is provided for convenience or for encoding agreement-term-specific options (*cf.* the use of the **Customisable** macro described earlier).

The most significant part of an agreement term are its guarantees, which come in two forms: guaranteed *states* and guaranteed *actions*. Formally, we first define an abstract **Guarantee*** type to capture the common attributes of both states and actions; namely, these are a reference to the actor obligated to ensure the guarantee is satisfied, and an optional **Constant*** (described in Section 5.2) serving as a domain-specific indication of the guarantee's priority:

> **Guarantee*** $\subset$ NamedEntity*
> *priority* $\subset$ Constant* $_{[0..1]}$
> *obligated* $\subset$ $\uparrow$Actor* $_{[1]}$

A guaranteed *state* describes some state of affairs that the obligated actor is responsible for maintaining. Typically, this will be a QoS constraint, such as *completion time of service X is less than 5 s* or *service X has greater than 90% availability*. We refer to this state of affairs as the guarantee's *post*-condition (since it represents the desired effect of the guarantee). To allow for multiple guaranteed states effective under different contingencies, an optional *pre*-condition is also permitted. Thus a guaranteed state is a **Guarantee*** with additional *pre* and *post* constraints:

> **State** $\subset$ Guarantee*
> *pre* $\subset$ Constraint* $_{[0..1]}$
> *post* $\subset$ Constraint* $_{[1]}$

A guaranteed *action*, instead, describes an obligation on an actor to perform (or refrain from performing) some specific action under specific conditions. Simple examples include obligations on the service provider to send periodic reports to the customer, or to pay penalties in the case of SLA violations. The description of a guaranteed action entails four elements:

- a 'policy' stating whether the action is mandatory, forbidden or optional,
- a specification of the (class of) events which *trigger* (or, depending on policy, *inhibit*) the action, referred to as the guaranteed action's *pre*-condition,
- a *time limit* within which the action must be performed (or during which the action is prohibited),
- a description of the action itself, which leads to the guaranteed action's *post*-condition,

The entity-type definition encapsulating this information is as follows:

> **Action** $\subset$ Guarantee*
> *policy* $\subset$ ENUM $_{[1]}$
> *pre* $\subset$ $\uparrow$EventClass* $_{[1]}$
> *limit* $\subset$ DURATION $_{[1]}$
> *post* $\subset$ ActionDef* $_{[1]}$

Formally, the action's *pre*-condition (trigger) is given as a *reference to* an **EventClass***, identifying a class of events. The SLA Model defines several classes of event, the simplest of which are **DATETIME** constants (i.e. time-stamps). Additional classes of event can be defined by domain-specific vocabularies (Section 6).

The action's *post*-condition, instead, is given as an **ActionDef***, which is essentially an empty placeholder to be filled by domain-specific action descriptions:

<u>**ActionDef***</u> $\subset$ E*

By way of illustration, the SLA Model defines an **ActionDef*** subtype representing a 'payment', i.e. a transfer of economic value. Since the actor obliged to make the payment is already given (see *Guarantee**), the formal definition of a payment need only identify the recipient and the value:

<u>**Payment**</u> $\subset$ ActionDef*
  recipient $\subset$ $\uparrow$Actor* $_{[1]}$
  value $\subset$ V* $_{[1]}$

Other action **ActionDef*** subtypes defined by the SLA Model are:

- **Invocation** : denoting the invocation of a specific interface operation,
- **Termination** : denoting the termination of an SLA,
- **Workflow** : denoting a composition of actions.

The example SLA Template presented in Section 7 illustrates the use of both guaranteed states and actions.

4 Interface Specifications

The functional capabilities of services are captured as functional interface specifications. The notion of 'interface' employed in the SLA Model is essentially a generalisation of WSDL 2.0, abstracting from web-service to 'service', and from the use of XML as concrete syntax. Accordingly, an interface is essentially a collection of named *operations*. For modularity, each interface specification may be a document in its own right, and interfaces may obtain specialisation hierarchies (i.e. extension, with operation inheritance). The **Interface** entity-type is defined as follows:

<u>**Interface**</u> $\subset$ Document*
  extended $\subset$ $\uparrow$Interface $_{[0+]}$
  operations $\subset$ Operation $_{[0+]}$

An interface operation is effected by a choreographed exchange of messages, specified by assigning appropriate message *types* to particular roles, or slots, in a standard exchange *pattern*. Potential faults (or exceptions) are specified in a similar fashion (we refer readers to the WSDL 2.0 specification for a more detailed explanation of these concepts). The **Operation** entity-type is defined as follows, with the value of the *message_label* attribute identifying the relevant pattern slot:

<u>**Operation**</u> $\subset$ Service*
pattern $\subset$ UUID $_{[1]}$
messages $\subset$ Message $_{[0+]}$
faults $\subset$ Message $_{[0+]}$

<u>**Message**</u> $\subset$ E*
message_label $\subset$ NAME $_{[1]}$
valuetype $\subset$ $\Uparrow$MessageType* $_{[1]}$

<u>**MessageType***</u> $\subset$ E*

<u>**Service***</u> $\subset$ NamedEntity*

Note that the **MessageType*** entity-type is defined as an empty specialisation of **E***, which means that messages may have arbitrary content. **MessageType*** subtypes are defined in vocabularies (Section 6) in just the same way that any domain-specific entity-type is defined (an example is given in Section 7).

The parent type of **Operation** is the abstract entity-type **Service***, which we first encountered in the previous section as the parent of **InterfaceDeclr**. To recap, an **InterfaceDeclr** comprises an **Interface**, which in turn comprises a set of **Operation**s. The **Service*** type can therefore be understood as encapsulating (through its subtypes) a collection of service **Operation**s.

In formal terms, an **Operation**, as we have just defined it, is essentially a prescription, or protocol, for exchanging messages. In contrast, when we speak of the *invocation* of an **Operation**, we are instead referring to the *execution* of this protocol; that is, we refer to a particular exchange of particular messages. In other words, an *invocation* is a specific *physical event* occurring at a specific point in space in time. Distinct invocations of the same **Operation** will thus have idiosyncratic properties (e.g. time and place) which are not represented at the level of protocol description. To describe such *event properties*, the SLA Model provides a dedicated **EventClass*** type (see also guaranteed actions in the previous section). For invocation events in particular, the model provides **InvocationClass***, the formal definition of which is as follows:

<u>**InvocationClass***</u> $\subset$ EventClass* $\Longleftrightarrow$ Service*
invocation_uuid $\subset$ UUID $_{[1]}$
request_time $\subset$ DATETIME $_{[1]}$
reply_time $\subset$ DATETIME $_{[1]}$
endpoint_uuid $\subset$ UUID $_{[1]}$
consumer_uuid $\subset$ UUID $_{[1]}$

The attributes of **InvocationClass*** denote properties of invocation events. The value of *request_time*, for example, gives the point in time at which an invocation request was received, while the value of *endpoint_uuid* identifies the endpoint at which the request was received. As such, it should be clear that tokens of the **InvocationClass*** type, or indeed of any **EventClass*** type, always constitute descriptions of *particular* events.

This, however, constitutes a problem. Since the purpose of a SLA(T) is to constrain *future* events (those constituting the service *to be* delivered), it is unlikely that **EventClass*** tokens will ever appear in SLA(T)s. Nevertheless, it is useful to refer to event properties. We may wish, for example, to define different QoS guarantees for a given service according to the *request_time*, or *endpoint_uuid* of invocations. To permit this, the SLA Model requires that each **EventClass*** type is associated with a corresponding entity-type. In the case of **InvocationClass***, the associated entity-type is **Service*** (indicated by the $\Longleftrightarrow$ symbol in the formal definition). This association permits **InvocationClass*** attributes to be referenced *as if* they were attributes of a **Service*** token.

This completes the SLA Model specification of **SLA**, **SLAT** and **Interface** document types. As stated in the introduction, all these definitions are *minimal*, encapsulating only the common, domain-independent content of SLA(T) documents. As we will see in Section 6, all the entity-types defined here may be arbitrarily extended by domain-specific vocabularies.

5 Value Types (the abstract constraint language)

The entity-type definitions presented in preceding sections made use of two important — but as yet undefined — expression types, namely: **Constraint*** (used for specifying QoS guarantees) and **Constant*** (the abstract supertype of all datatypes). These types both specialise the high-level type **V*** $\subset$ **L***, which denotes a class of 'value types'. A third value-type, thus far unmentioned, is **Parametric** $\subset$ **V***, denoting an extensible set of expressions with a parametric, or functional, form. Examples include arithmetic and set operators ($+$, $\times$, $\subset$, $\in$, etc), and QoS 'metrics' (e.g. *completion_time*, *arrival_rate*, *availability*, etc). Taken together, these types constitute an *abstract constraint language*, which we describe in the following subsections.

5.1 Constraint Expressions

The starting point in the abstract constraint language, is the **Constraint*** expression type. A constraint expression is some statement, or formula, which places bounds on the permitted value of some variable. A constraint may be *atomic* or *compound*. Examples of atomic constraints include the following:

- $X < 4$,
- $X + Y >= Z$,
- $foo(Y) \; != goo(Z)$,
- $Z \in \{ a, b, c \}$,

- *completion_time*(S) $<$ 10 s,

In general, an atomic constraint could be defined as an ordered relation between a variable and a value. The expression ‘X $<$ 4’, for example, would comprise the relation ‘$<$’ between values ‘X’ and ‘4’. In the SLA Model, however, we take a slightly more convoluted approach, and define an atomic constraint as a variable (e.g. ‘X’) bound to lie in some *domain* (e.g. ‘$<$ 4’). This approach allows the domain part of the expression to be employed independently of constraints.

A compound constraint is some logical combination of *sub*-constraints. For maximum flexibility we also allow both atomic and compound domains, where a compound domain is some logical combination of *sub*-domains (e.g. the conjunction ‘$>$ 4 and $<$ 10’).

To model constraints, we first introduce the following abstract types:

Constraint* $\subset$ **V*** : the abstract supertype of constraints,

Domain* $\subset$ **V*** : the abstract supertype of domains,

Constant* $\subset$ **V*** : the abstract supertype of constants,

The concrete atomic and compound versions of constraints and domains are then given by the following expression type definitions:

AtomicConstraint $\subset$ **Constraint*** : each token is an ordered pair <***c*,*d***>, where ***c*** is a non-empty array (an ordered list) of **Constant*** values, each member of which is constrained to lie in the domain ***d*** $\in$ **Domain***.

CompoundConstraint $\subset$ **Constraint*** : each token is an ordered pair <***o*,*C***>, where ***o*** $\in$ **UUID** uniquely identifies a compound operator (e.g. ‘and’, ‘or’, or ‘not’), and ***C*** $\subset$ **Constraint*** is an unordered set of *sub*-constraints.

AtomicDomain $\subset$ **Domain*** : each token is an ordered pair <***o*,*c***>, where ***o*** $\in$ **UUID** uniquely identifies a domain operator (e.g. $<$, $>=$, $!=$, etc), and ***c*** $\in$ **Constant*** specifies a domain boundary (according to the semantics of the domain operator).

CompoundDomain $\subset$ **Domain*** : each token is an ordered pair <***o*,*D***>, where ***o*** $\in$ **UUID** uniquely identifies a compound operator (e.g. ‘and’, ‘or’, or ‘not’), and ***D*** $\subset$ **Domain*** is an unordered set of *sub*-domains.

Note that the constrained variable in an **AtomicConstraint** is defined as an array. So, for example, the constraint ‘X $<$ 4’ would represent ‘[X] $<$ 4’ (where ‘[..]’ denotes an array). Semantically, a constraint such as ‘[X,Y] $<$ 4’ could equally be expressed as a conjunction ‘(X $<$ 4) and (Y $<$ 4)’. Constraints are defined in this way to ensure a consistent semantics for **EventClass*** types (Section 4), a full discussion of which is beyond the scope of this chapter.

The SLA Model predefines several domain operators, namely: the standard comparison operators ($<$, $>$, $<=$, $>=$, $=$ and $!=$), a ‘matches’ operator for comparing character strings against regular expressions, and a set membership operator (‘member_of’). Three compound (logical) operators are also defined: ‘and’ (conjunction), ‘or’ (disjunction) and ‘not’ (negation). If required, additional, domain-specific operators can be specified using vocabularies, as described in Section 6.

5.2 Constants and Datatypes

Constant expressions, encapsulated by the abstract type **Constant***, are the most primitive expressions of the SLA Model, constituting the terminal nodes in document content hierarchy. Constant expressions include such things as metric quantities (e.g. '4 s', '10 MB', '90%', etc.), e-mail and web addresses (e.g. 'http://sla-at-soi.eu/'), Boolean values (e.g. 'true' and 'false'), time-stamps (e.g. 'Wed Dec 15 18:38:0.0 CET'), and others. The term *datatype* is used informally to refer to any subtype of **Constant***.

We have already encountered some of the datatypes built-in to the SLA Model. The complete list is as follows:

- **TEXT** : opaque (unparsed) character strings,
- **REGEX** : regular expressions (explained below),
- **ENUM** : enumerations (e.g. Section 3.1),
- **PATH** : typically takes the form of a navigable route through the document hierarchy, identifying some target expression token,
- **UUID** : a universally unique identifier (e.g. a URI),
- **NAME** : used in particular as the name of a **NamedEntity***,
- **CARD** : cardinality constraints (e.g. '0..1', '1+'), etc.
- **BOOL** : Boolean values,
- **STND** : standard forms (explained below),
- **NUMERIC*** : abstract supertype of numeric quantities.

The **NUMERIC*** datatype is an abstract supertype encapsulating numeric constants, and, in particular, metric quantities. The SLA Model provides the following built-in specialisations:

- **QUANTITY** : non-metric real values, e.g. '1.435', 'pi',
- **COUNT** : non-metric integer values,
- **PERCENT** : percentiles, e.g. '90%',
- **DURATION** : periods of time, e.g. '4 s', '2 days',
- **CURRENCY** : e.g. '10 Euros',
- **DATASIZE** : e.g. '5 bytes', '100 GB',
- **DATARATE** : e.g. '1 GB_per_s' (gigabytes per second),
- **TXRATE** : transaction rates, e.g. '2 tx_per_day' (transactions per day),
- **LENGTH** : e.g. '4 m', '10 cm',
- **AREA** : e.g. '10 m2' (metres squared),
- **FREQUENCY** : e.g. '200 Hz', '33 rpm',
- **WEIGHT** : e.g. '25 kg',
- **POWER** : e.g. '300 mW',
- **ENERGY** : e.g. '37 KWh',

To compare and validate constant expressions, we require a means to determine the datatype of any given constant token. To determine that the phrase '4 kg $<$ 10 J' is invalid, for example, we need to know that '4 kg' and '10 J' denote measures with different (and incomparable) datatypes (**WEIGHT** and **ENERGY** *respectively*). To achieve this, datatypes can be associated with regular expressions, which constrains the format of tokens, and allows for the determination of type by pattern matching. The **WEIGHT** datatype, for example, has an associated regular expression '[x] kg', where '[x]' is interpreted as a placeholder for a number, such that any character sequence matching '[x] kg' will be interpreted as a **WEIGHT** token. The built-in **REGEX** datatype denotes the class of such regular expressions.

The **STND** datatype extends this use of regular expressions to also allow definitions of data-conversion formula. The **REGEX** token '[x] hrs', for example, is mapped onto the **STND** token '[x*3600] s' (which is referred to as its *standard form*), and serves to encode the formula required to convert a duration expressed in hours into the equivalent duration expressed in seconds.

All the datatypes listed above are defined as part of the SLA Model. In Section 6 we will see how vocabularies can be used to define additional datatypes to meet domain-specific requirements.

5.3 Parametrics

The third, and final class of value tokens is the **Parametric** $\subset$ **V*** type, denoting expressions which have a parametric, or functional, form. Common examples include:

- arithmetic operations, e.g. '$X + 4$', '8×12',
- aggregate operations, e.g. '*sum*([1,2,3])', '*mean*([4,5,6])'
- set operators, e.g. '$X \in [a,b,c] \cup [d,e,f]$',
- QoS metrics, e.g. '*completion_time*(*S*)', where *S* denotes a set of service invocations.

The formal definition of **Parametric** type is as follows:

Parametric $\subset$ **V*** : each token is an ordered pair <***f***,***P***>, where ***f*** uniquely identifies an operator (i.e. a 'function' or 'predicate' name), and ***P*** $\subset$ **V*** is an ordered set of parameters.

For validation purposes, we also need a means to specify, for each function name (i.e. f in the preceding definition) the required arity and types of its parameters. In addition, **Parametric** expressions have the special property that, as well as conforming to a syntactic type, they also obtain a *semantic 'role'*, which is defined as the syntactic type to which the expression *evaluates* when interpreted. The token '*sum*([2 mins, 3 s])', for example, denotes the summation over the durations '2 mins' and '3 s' which evaluates to the single duration '123 s'. The semantic role of the token '*sum*([2 mins, 3 s])' is the type of this evaluated result, namely, the datatype **DURATION**. The significance of the semantic role is that **Parametric** expressions may be used anywhere that tokens with their semantic role are permitted. If a **DURATION** constant is required, for example, then *any* **Parametric** expression which evaluates to a **DURATION** constant may be used instead.

The SLA Model allows all this information to be captured formally in vocabularies (Section 6). The *sum* operator, for example, is formally defined as a single non-empty array of some numeric type **N** $\subset$ **NUMERIC** as parameter, and as evaluating to a single value of the same type. We can express this concisely with the following notation:

- $sum(\mathbf{N}_{[1+]}) \rightarrow \mathbf{N}_{[1]} \subset \mathbf{NUMERIC}$.

The SLA Model defines many built-in **Parametric*** types covering, among others, the common arithmetic, aggregation and set operators as well as QoS metrics. A complete description of all the parametric types would require more space than is available here. To give some flavour of the model, however, the following is a complete list of formal definitions for the built-in QoS metrics:

- *accessibility*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **QUANTITY**$_{[0+]}$.
- *arrival_rate*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **TXRATE**$_{[0+]}$.
- *availability*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **QUANTITY**$_{[0+]}$.
- *completion_time*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **DURATION**$_{[0+]}$.
- *isolation*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **BOOL**$_{[0+]}$.
- *mttf*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **DURATION**$_{[1]}$.
- *mttr*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **DURATION**$_{[1]}$.
- *non_repudiation*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **TEXT**$_{[1]}$.
- *regulatory*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **TEXT**$_{[1+]}$.
- *supported_standards*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **TEXT**$_{[1+]}$.
- *throughput*($\uparrow$**InvocationClass***$_{[1]}$) $\rightarrow$ **TXRATE**$_{[1]}$.

To close this section, we should mention two additional **Parametric** types that will be used in the example SLAT in Section 7:

- *violation*($\uparrow$**Guarantee***$_{[1]}$) $\rightarrow$ **E**$_{[1]}$ $\subset$ $\uparrow$**EventClass***.
- *union*(**E**$_{[2+]}$) $\rightarrow$ **E**$_{[1]}$ $\subset$ $\uparrow$**EventClass***.

The first of these, *violation*, is used to specify a class of events whose members are the individual occurrences of the violation of some guarantee. The second, *union*, serves to combine diverse classes of event into a single event class. By combining the two, we can specify a class of events whose members are the occurrences of violations of any of a given set of guarantees.

Additional domain-specific **Parametric** types can be specified using vocabularies, which we describe in the next section.

6 Domain-Specific Vocabularies

The previous sections outlined the basic content of the SLA Model, which, by way of summary, comprises an abstract constraint language (Section 5), a document model for **Interface** specifications (Section 4), and, building on these, document models for SLAs and SLATs (Section 3). Many aspects of the SLA Model, however, are open and extensible, supporting customisation to domain-specific requirements. Extensions to the model are specified using vocabularies, which we describe in this section.

A vocabulary is a document comprising a list of vocabulary terms, each of which specifies a particular extension to the SLA Model. Formally, a vocabulary is encapsulated by the entity-type **Vocabulary**:

Vocabulary $\subset$ Document*

terms $\subset$ Term* $_{[1+]}$

The type **Term*** is the abstract supertype of all vocabulary terms, of which there are seven concrete specialisations, each serving a different purpose. For reasons of space we can not present their complete formal definitions, but the following list provides brief informal descriptions:

Term* $\subset$ **E*** : abstract supertype of vocabulary terms.

EntityType $\subset$ **Term*** : each token provides a formal definition of an entity-type (i.e. a subtype of **E***). All the formal entity-type definitions provided in this chapter — for example, the **Vocabulary** definition above — are perfectly valid examples of **EntityType** tokens.

DataType $\subset$ **Term*** : each token provides a formal definition of a datatype (i.e. a subtype of **Constant***), which comprises a unique identifier (**UUID**) and supertype.

DataValue $\subset$ **Term*** : each token associates a datatype with a regular expression (**REGEX**) and optional standard form (**STND**), the purposes of which are explained in Section 5.2.

ParametricType $\subset$ **Term*** : each token specifies a parametric operator (**UUID**), together with its required arity and parameter types, and its semantic role. The 'QoS metrics' listed in Section 5.3 are all valid examples of **ParametricType** tokens.

DomainOp $\subset$ **ParametricType** : each token specifies a domain operator (*cf* the definition of **AtomicDomain** in section 5.1).

CompoundOp $\subset$ **Term*** : each token specifies a compound operator (*cf.* the definitions of **CompoundConstraint** and **CompoundDomain** in Section 5.1).

EventClass* $\subset$ **Term*** : each token specifies a class of events, defining a unique identifier (**UUID**) for the class, the entity-type with which it is associated, and a list of monitorable attributes. The **InvocationClass*** defined in Section 4 is a valid example of an **EventClass*** token.

Vocabulary documents thus allow for a considerable degree of domain-specific customisation, supporting the definition of new entity-types, datatypes and data-formats, parametric, domain and compound operators, and classes of event. Domain-specific applications may pick and choose from existing vocabularies, or create entirely new ones, as per their needs, thus supporting a modular approach to development. Individual vocabularies are identified by a URI, which also constitutes a *namespace* (in the manner of XML) for the terms defined in that vocabulary.

Thus the SLA Model itself can in large part be specified using vocabularies. The SLA Model is specified in four distinct parts: The first, referred to as the 'core', comprises all the basic definitions given in Section 2, the abstract constraint language (Section 5), and the definition of vocabulary documents (this section). Interface specifications (Section 4), SLAs and SLATs (Section 3), and QoS Metrics (Section 5.3) are then each specified in distinct vocabularies. The namespace URIs of these vocabularies are as follows:

- *Core* : http://www.slaatsoi.org/coremodel#
- *Interfaces* : http://www.slaatsoi.org/interfaces#
- *SLA(T)s* : http://www.slaatsoi.org/slamodel#
- *QoS Metrics* : http://www.slaatsoi.org/commonTerms#

For simplicity, we have until now ignored these namespace URIs. It should be borne in mind, however, that all the expression types defined by the SLA Model are formally identified by URIs. The formal identifier for the **NamedEntity*** entity-type, to take a random example, is the URI http://www.slaatsoi.org/coremodel#NamedEntity.

In the final section, below, we provide an example SLAT which illustrates how the SLA Model is applied, and how diverse vocabularies work together.

7 An Example SLA

We close this chapter on the SLA Model with a concrete example of an SLA Template. Since the SLA Model is an *abstract* syntax, the first task is to choose an appropriate concrete syntax for the example. For simplicity, we will use XML[1], assuming that it is familiar to most readers. Line numbers are added to facilitate description. The content of the SLAT will be described as the example progresses.

We start by describing the service that is the subject of the SLAT. Since our focus is the SLAT itself, we will keep the service simple and intuitive: a *product purchasing service* comprising a single operation, 'BuyProduct', offered by a provider 'Fred'. The interface for the service is specified as an *interface document*, i.e., an instance (token) of the entity-type **Interface** (described in Section 4). The complete document is as follows:

```
 1: <iface:Interface
 2:  xmlns:iface = "http://www.slaatsoi.org/interfaces#"
 3: >
 4:   <vocabularies>
 5:     http://www.fred.com/freds_vocab
 6:   </vocabularies>
 7:   <operations>
 8:     <iface:Operation>
 9:       <name>BuyProduct</name>
10:       <pattern>http://www.w3.org/ns/wsdl/in-out</pattern>
11:       <messages>
12:         <iface:Message>
13:           <message_label>In</message_label>
14:           <valuetype>
15:             http://www.fred.com/freds_vocab#BuyProduct.In
16:           </valuetype>
17:         </iface:Message>
18:       </messages>
19:     </iface:Operation>
20:   </operations>
21: </iface:Interface>
```

[1] For reasons of space we do not provide an XML Schema. The mapping from the abstract syntax to XML should be, however, self-evident.

The opening element (lines 1–3) announces the document to be an instance (token) of the entity-type **iface:Interface**, where ‘iface’ denotes the URI namespace ‘http://www.slaatsoi.org/interfaces#’, defined by the SLA Model for interface document terms. The first child element (lines 4–6) lists the various vocabularies against which the document content must be validated. In this case, just one vocabulary is used (available at the URI ‘http://www.fred.com/freds_vocab’), which we will describe shortly.

The remaining content (lines 7–20) defines an interface operation with the name ‘BuyProduct’ (line 9), and standard ‘in-out’ messaging pattern, as identified by the URI ‘http://www.w3.org/ns/wsdl/in-out’ (line 10). Lines 12–17 then assign a message-type, identified as ‘http://www.fred.com/freds_vocab#BuyProduct.In’ (line 15), to the pattern role ‘In’ (line 13). For modularity, the message type is defined in the imported domain-specific vocabulary (line 5). This vocabulary is a distinct document, whose content is as follows:

```
 1: <core:Vocabulary
 2:   xmlns:core = "http://www.slaatsoi.org/coremodel#"
 3:   xmlns:iface = "http://www.slaatsoi.org/interfaces#"
 4: >
 5:   <vocabularies>
 6:     http://www.slaatsoi.org/interfaces
 7:   </vocabularies>
 8:   <terms>
 9:    <core:EntityType>
10:       <uuid>
11:          http://www.fred.com/freds_vocab#BuyProduct.In
12:       </uuid>
13:       <supertype>
14:          http://www.slaatsoi.org/interfaces#MessageType
15:       </supertype>
16:       <concrete>yes</concrete>
17:       <definition>
18:         defines the 'In' message of 'BuyProduct'
19:       </definition>
20:       <attributeTypes>
21:         <core:AttributeType>
22:           <name>product_id</name>
23:           <valuetype>
24:              http://www.slaatsoi.org/coremodel#TEXT
25:           </valuetype>
26:           <cardinality>1</cardinality>
27:           <definition>
28:             identifies the product to buy
29:           </definition>
30:         </core:AttributeType>
31:       </attributeTypes>
32:     </core:EntityType>
33:   </terms>
34: </core:Vocabulary>
```

As before, the opening element announces the document entity-type, which is now **core:Vocabulary**, with ‘core’ denoting ‘http://www.slaatsoi.org/coremodel#’, the URI namespace of the core SLA Model terms. Since the purpose of this vocabulary is to define the message-type used by the ‘BuyProduct’ operation, we first need to import (in lines 5–7) the ‘http://www.slaatsoi.org/interfaces’ vocabulary in which ‘iface:MessageType’ is defined (the core vocabulary is imported automatically and does not need to be included). Vocabulary imports are transitive in the SLA Model, which means that the interfaces vocabulary is also automatically available to the interface specification document.

The message-type required for the interface is specified using an **core:EntityType** vocabulary term (lines 9–32). This term defines a new concrete (line 16) subtype of **iface:MessageType** (line 13), ‘http://www.fred.com/freds_vocab#BuyProduct.In’ (line 11), whose purpose is described in the scope-note (lines 17–19). It has a single attribute, defined in lines 20–31, with the name ‘product_id’ (line 22), whose value is a single (line 23) opaque character string (datatype **core:TEXT**; line 24). Using the notation introduced in Section 2, we would write this entity-type definition as:

$$\underline{\textbf{http://www.fred.com/freds\_vocab\#BuyProduct.In}} \subset \text{iface:MessageType*}$$
$$\quad \textit{product\_id} \subset \text{core:TEXT}_{[1]}$$

These two documents fully specify the service interface. The last step is to create an SLAT to specify quality constraints and party obligations in respect of this service.

In outline, the SLAT will provide customers the option of two ‘service levels’: *basic* and *premium*. At the *basic* level, the customer is guaranteed a completion time for service invocations of less than 2 hours, while at the *premium* level, this is improved to less than 30 minutes. Each time a guarantee is violated, the provider, ‘Fred’, is given two weeks to pay a penalty of 10 Euros. The complete SLAT is given by the remaining XML listings below, which for ease of description we will explain section by section.

The opening XML elements are straightforward, announcing that the document is an SLAT, and enumerating namespace abbreviations. In addition, for convenience only, we have also added XML entity declarations (lines 1–6) denoting the core, interface, SLA(T) and QoS Metric URIs. The SLAT also needs to explicitly import the SLA(T) and QoS Metric vocabularies (lines 13–16).

```
 1: <!DOCTYPE E [",
 2: <!ENTITY core "http://www.slaatsoi.org/coremodel#">
 3: <!ENTITY iface "http://www.slaatsoi.org/interfaces#">
 4: <!ENTITY sla "http://www.slaatsoi.org/slamodel#">
 5: <!ENTITY qos "http://www.slaatsoi.org/commonTerms#">
 6: ]>",
 7: <sla:SLAT>
 8:   xmlns:core = "&core;"
 9:   xmlns:iface = "&iface;"
10:   xmlns:sla = "&sla;"
11:   xmlns:qos = "&qos;"
12: >
13:   <vocabularies>
```

```
14:     <item>http://www.slaatsoi.org/commonTerms</item>
15:     <item>http://www.slaatsoi.org/slamodel</item>
16:   </vocabularies>
```

The first content proper of the SLAT is a *parties* section (lines 17–26), which in this case distinguishes just two SLA actors: the provider, ‘Fred’, and customer, ‘TheCustomer’. Note that the SLA Model requires merely that relevant parties are distinguished and assigned SLA(T) roles. Additional party information can be included, but is treated as domain-specific; that is, additional party information needs to be specified by domain-specific extensions to the basic SLA(T) document definition.

```
17:   <parties>
18:     <sla:Party>
19:       <name>Fred</name>
20:       <role>provider</role>
21:     </sla:Party>
22:     <sla:Party>
23:       <name>TheCustomer</name>
24:       <role>customer</role>
25:     </sla:Party>
26:   </parties>
```

Having identified the key actors, we next declare (in lines 27–43) all the service interface(s) which are the subject of the SLAT. In this case, there is only the *product purchase interface* defined earlier, whose interface specification document we will reference (line 40) using the URI ‘http://www.fred.com/freds_service’. Note, however, that the use of a URI here is not obligated by the SLA Model. References may take any form, and the mechanism(s) by which references are resolved is application-specific. The SLA@SOI implementation assumes the use of URIs mapped to URLs.

The **sla:InterfaceDeclr** entity token specifies that this interface is to be provided by ‘Fred’ (line 30), that the intended consumer is ‘TheCustomer’ (line 31), and that it is accessible only by ‘e-mail’ (line 36) at the address ‘fred@xyz.com’ (line 35). We employ an e-mail protocol here for no other reason than to emphasise that the SLA Model is not restricted to standard web-service protocols. For internal reference, both the **sla:InterfaceDeclr** and **sla:Endpoint** token are assigned identifiers: ‘IF1’ and ‘EPR1’ (*resp.*).

```
27:   <interfaceDeclrs>
28:     <sla:InterfaceDeclr>
29:       <name>IF1</name>
30:       <provider>Fred</provider>
31:       <consumers>TheCustomer</consumers>
32:       <endpoints>
33:         <sla:Endpoint>
34:           <name>EPR1</name>
35:           <location>fred@xyz.com</location>
36:           <protocol>e-mail</protocol>
37:         </sla:Endpoint>
38:       </endpoints>
```

```
39:       <interface>
40:         http://www.fred.com/freds_service
41:       </interface>
42:     </sla:InterfaceDeclr>
43:   </interfaceDeclrs>
```

In the next section, *macros* (lines 44–63), we introduce the ‘service level’ options together with any other macros that may be useful. The ‘service level’ options are encoded in lines 45–56 as a **sla:Customisable** macro ‘X’ (line 46), denoting an expression whose value must be either ‘premium’ (line 51) or ‘basic’ (line 52), with ‘premium’ as the default option (line 47). For convenience, we also define a second macro ‘S’ (line 58), denoting the expression ‘IF1/interface[0]/BuyProduct’ (line 60). This expression is a **core:PATH** token which resolves to the ‘BuyProduct’ **iface:Operation** entity in the embedded interface document. (As with all references, the particular format of the path is application-specific.) As such, the value ‘S’ can from now on be used to refer to the ‘BuyProduct’ operation.

```
44:   <macros>
45:     <sla:Customisable>
46:       <name>X</name>
47:       <expression>premium</expression>
48:      <domain>
49:         <core:AtomicDomain op="&core;member_of">",
50:           <array>
51:             <item>premium</item>
52:             <item>basic</item>
53:           </array>
54:         </core:AtomicDomain>
55:       </domain>
56:     <sla:Customisable>
57:     <core:Macro>
58:       <name>S</name>
59:       <expression>
60:         IF1/interface[0]/BuyProduct
61:       </expression>
62:     </core:Macro>
63:   </macros>
```

The final section of the SLAT details the agreement terms. For the present example, there is only one agreement term, given the name ‘AT1’. The opening elements are as follows (lines 64–67):

```
64:   <agreementTerms>
65:     <sla:AgreementTerm>
66:       <name>AT1</name>
67:       <guarantees>
```

The required completion time and penalty guarantees (see above) will be encoded as two guaranteed states and a guaranteed action, named ‘G1’, ‘G2’ and ‘G3’ (*respectively*). The first guaranteed state (lines 68–89) encodes an obligation on ‘Fred’ (line 70) to ensure that, in the case that the ‘basic’ service level is selected, the completion time of any invocation of the ‘BuyProduct’ operation (line 82) is less than

2 hours (line 85). In a more concise form, we may express this guarantee as the following rule: *if X = 'basic', then completion_time(S) < 2 hrs*.

```
 68:         <sla:State>
 69:           <name>G1</name>
 70:           <obligated>Fred</obligated>
 71:           <pre>
 72:             <core:AtomicConstraint>
 73:               <item>X</item>
 74:               <core:AtomicDomain op="&core;equals">
 75:                  basic
 76:               </core:AtomicDomain>
 77:             </core:AtomicConstraint>
 78:           </pre>
 79:           <post>
 80:             <core:AtomicConstraint>
 81:               <core:Parametric op="&qos;completion_time">
 82:                 S
 83:               </core:Parametric>
 84:               <core:AtomicDomain op="&core;less_than">
 85:                 2 hrs
 86:               </core:AtomicDomain>
 87:             </core:AtomicConstraint>
 88:           </post>
 89:         </sla:State>
```

In the same manner, the second guaranteed state (lines 90–111) encodes the following rule: *if X = 'premium', then completion_time(S) < 30 mins*.

```
 90:         <sla:State>
 91:           <name>G2</name>
 92:           <obligated>Fred</obligated>
 93:           <pre>
 94:             <core:AtomicConstraint>
 95:               <item>X</item>
 96:               <core:AtomicDomain op="&core;equals">
 97:                 premium
 98:               </core:AtomicDomain>
 99:             </core:AtomicConstraint>
100:           </pre>
101:           <post>
102:             <core:AtomicConstraint>
103:               <core:Parametric op="&qos;completion_time">
104:                 S
105:               </core:Parametric>
106:               <core:AtomicDomain op="&core;less_than">
107:                 30 mins
108:               </core:AtomicDomain>
109:             </core:AtomicConstraint>
110:           </post>
111:         </sla:State>
```

The third and final guarantee encodes the penalty action. The trigger (precondition) for the action (lines 112–135) is the occurrence of a violation of either of the guaranteed states ‘G1’ and ‘G2’ (Section 5.3 for an explanation of the *union* and *violation* parametrics). The guarantee specifies that, in case of such a violation, there is a ‘mandatory’ (line 115) obligation on ‘Fred’ (line 114) to make a payment of ‘10 Euros’ (line 1132) to ‘TheCustomer’ (line 131), with a payment deadline of ‘2 weeks’ (line 128) from the violation trigger event. The guarantee is violated if ‘Fred’ fails to make this payment within this time-frame.

```
112:          <sla:Action>
113:            <name>G3</name>
114:            <obligated>Fred</obligated>
115:            <policy>mandatory</policy>
116:            <pre>
117:              <core:Parametric op="&core;union">
118:                <array>
119:                  <core:Parametric op="&sla;violation">
120:                    G1
121:                  </core:Parametric>
122:                  <core:Parametric op="&sla;violation">
123:                    G2
124:                  </core:Parametric>
125:                </array>
126:              </core:Parametric>
127:            </pre>
128:            <limit>2 weeks</limit>
129:            <post>
130:              <sla:Payment>
131:                <recipient>TheCustomer</recipient>
132:                <value>10 Euros</value>
133:              </sla:Payment>
134:            </post>
135:          </sla:Action>
```

The remaining lines of XML (lines 136–139) close the agreement terms section, and complete the SLAT.

```
136:        </guarantees>
137:      </sla:AgreementTerm>
138:    </agreementTerms>
139: </sla:SLAT>
```

To convert this SLA Template into an SLA, we just need to add values for the mandatory SLA attributes *agreedAt*, *effectiveFrom* and *effectiveUntil*.

8 Conclusion

The SLA model meets the project requirements and has been tested in practical application. The model offers a language-independent specification of SLA(T) content at a fine-grained level of detail, which is both highly expressive and inherently extensible. The model has been applied to the business use cases of the SLA@SOI project (see also Chapter 'Introduction to the SLA@SOI Industrial Use Cases') and is already used by a number of European projects, for example Contrail[2].

References

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, Web services agreement specification (ws-agreement). Grid Forum Document GFD.107, The Open Grid Forum, Joliet, Illinois, United States, 2007

[2] A. Keller and H. Ludwig, The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, Web Services Description Language (WSDL) 1.1 W3C Note, World Wide Web Consortium, 15 March 2001

[2] Contrail – Open Computing Infrastructures for Elastic Services: http://contrail-project.eu